Coding Recommendations for Licensees

Palm OS 5.0

CONTRIBUTORS

Written by Clif Liu
Engineering contributions by Steve Minns, Jesse Donaldson, Noah Gibbs, and Lee Taylor.

Coding Recommendations for Licensees
Document Number 5030-001
April 12, 2002

PalmSource, Inc.
5470 Great America Pkwy.
Santa Clara, CA 95054
USA
www.palmos.com

Table of Contents

About This Document

This document contains a set of recommendations that make the code you write easier for other developers to maintain. It is not the *Palm™ Coding Standards and Guidelines*, a document currently under revision that defines the coding standards that all developers at Palm are expected to abide by. This document was created for the interim to help licensees develop code that Palm can use with a minimum of maintenance and debugging hassles.

The information presented here is fairly general. With few exceptions, it doesn't dictate a set of rules that you have to follow. Rather, it suggests ways of solving common development problems. But if you follow these recommendations, you will be rewarded with cleaner code, shorter debugging cycles, and most importantly, the deep gratitude of those who inherit your code.

Additional Resources

- Documentation

  Palm publishes its latest versions of this and other documents for Palm OS developers at

  http://www.palmos.com/dev/support/docs/

- Training

  Palm and its partners host training classes for Palm OS developers. For topics and schedules, check

  http://www.palmos.com/dev/training

- Knowledge Base

  The Knowledge Base is a fast, web-based database of technical information. Search for frequently asked questions

(FAQs), sample code, white papers, and the development documentation at

http://www.palmos.com/dev/support/kb/

1

General Recommendations

Factoring Code

Common code that appears in multiple places is hard to maintain. If it needs to be changed, you must change it everywhere it appears. If you miss one of these changes, you introduce a bug. *Factoring* is a generic term that means eliminating repeated code. The word comes from algebra. On the left-hand side of the equation below, *a+b* appears twice, but on the factored right-hand side, it appears only once.

$$(a + b)c + (a + b)d = (a + b)(c + d)$$

Similarly, if the same code appears in two places, you can and should rewrite it so it appears only once. One way to do this is to create a function that performs the common code.

Code Factoring Example

An example where factoring is useful occurs commonly in Palm application programming. Given a resource ID, one often needs to determine the pointer to the corresponding user interface element in the current form using the code in Listing 1.1.

Listing 1.1 An often-used code snippet

```
FormType *formP = FrmGetActiveForm();
objectP = FrmGetObjectPtr(formP,
  FrmGetObjectIndex(formP, objectID));
```

In the following example, this code appears twice:

Listing 1.2 Code before factoring

```
static Boolean MainFormHandleEvent(EventType *eventP)
{
  FormType *mainFormP;
  FieldType *timeIntervalFieldP;

  // CODE
  // ...

  switch (eventP->eType) {
    case frmOpenEvent:
      mainFormP = FrmGetActiveForm();
      MainFormInit(mainFormP);
      FrmDrawForm(mainFormP);

      // CODE
      // ...

      // Get a pointer to the time interval text field
      timeIntervalFieldP = FrmGetObjectPtr(mainFormP,
        FrmGetObjectIndex(mainFormP,MainIntervalField));

      // Display time interval in the text field
      break;

    case ctlSelectEvent:
      switch (eventP->data.ctlSelect.controlID) {
        case MainStartPushButton:
        {
          Int32 timeInterval;

          mainFormP = FrmGetActiveForm();
          timeIntervalFieldP =
            FrmGetObjectPtr(mainFormP,
            FrmGetObjectIndex(mainFormP,
            MainIntervalField));
          timeInterval =
            StrAToI(FldGetTextPtr(timeIntervalFieldP));

          // CODE
          // ...

          break;
        }

        // CODE
        // ...
      } // switch (eventP->data.ctlSelect.controlID)

      // CODE
      // ...

   } // switch (eventP->eType);
} // MainFormHandleEvent
```

In both cases, the code sets the value of `timeIntervalFieldP`. To eliminate the repeated code, you define a function that encapsulates the snippet and call it from the main code as shown in Listing 1.3.

Listing 1.3 The factored code

```
void *GetObjectPtr(UInt16 objectID)
{
   FormType *activeFormP = FrmGetActiveForm();
   return FrmGetObjectPtr(activeFormP,
      FrmGetObjectIndex(activeFormP, objectID));
}


static Boolean MainFormHandleEvent(EventType *eventP)
{
   FormType *mainFormP;
   FieldType *timeIntervalFieldP;

   // CODE
   // ...

   switch (eventP->eType) {
      case frmOpenEvent:
         mainFormP = FrmGetActiveForm();
         MainFormInit(mainFormP);
         FrmDrawForm(mainFormP);

         // CODE
         // ...

         // Get a pointer to the time interval text field
         timeIntervalFieldP =
            GetObjectPtr(MainIntervalField);

         // Display time interval in the text field
```

```
          // CODE
          // ...

          break;

        case ctlSelectEvent:
          switch (eventP->data.ctlSelect.controlID) {
            case MainStartPushButton:
            {
              Int32 timeInterval;

              timeIntervalFieldP =
                GetObjectPtr(MainIntervalField);
              timeInterval =
                StrAToI(FldGetTextPtr(timeIntervalFieldP));

              // CODE
              // ...

              break;
            }

            // CODE
            // ...

          } // switch(eventP->data.ctlSelect.controlID)

        // CODE
        // ...

    } // switch(eventP->eType);
} // MainFormHandleEvent
```

Recommendations

- In general, the same code should not appear in more than one place.
- Don’t assume that if a code segment is very short that it’s easier to repeat it. Specifically, if the code will likely need to be modified later, you should factor it regardless of how short it is.
- From the beginning of the design process, organize your code so it consists of small reusable segments.

Length of Functions

Long functions are difficult to read, understand, and maintain. You should aim to have your functions no longer than 60 lines, the amount of text that fits on one single-spaced page. Don't count the comment header in the 60 lines.

To cut a long function down, determine which pieces of the function are independent of each other, encapsulate them within functions, and call the component functions from the main function. Be sure you name the component functions well and comment them. Naming is discussed in "Naming" on page 5, and comments are discussed in "Comments" on page 9.

Don't go overboard in splitting your function. Every function you create adds complexity to your code and makes it harder to maintain, even though it simplifies the original long function. Strike a balance between making your functions too long and too short.

Sometimes you will find that breaking down a long function creates opportunities to factor your code. Remember that factoring your code almost always improves it.

Not all functions can be shortened to 60 lines. For example, Listing 1.3 on page 3 shows an event handling function that consists of a switch statement. The code associated with the switch construct alone is more than 60 lines long.

Recommendation

Shorten functions longer than 60 lines, if possible. Do not count the comment header in the 60 lines. At the same time, be sure that the component functions are large enough that it's worthwhile to add them. This depends on the complexity of the tasks they perform.

Naming

When you choose good names for your variables, functions, preprocessor identifiers, types, files, and so on, you make your code much easier to understand and maintain. Good names are readable and self-explanatory. Ideally, a developer can read a name and understand what the name refers to—without looking at the code.

Traditionally, some developers choose short names for their variables, for example, `a`, `i`, and `j`. The name `i` provides only a little information. It indicates that the variable is probably an index variable. A name like `addressRecordIndex` is far more revealing. Avoid short obscure names, even if they save you some typing effort. Remember that the time that people spend reading your code far exceeds the amount of time you spend writing it. In addition, modern text editors provide cut and paste operations that save you the tedium of typing and retyping long names.

Table 1.1 Examples of good names

`timeIntervalFieldID`
`defaultTimeInterval`
`gAddressRecordH`
`addressRecordIndex`

Table 1.2 Some examples of names to avoid

`a`
`foo`
`mpDnLnRgCh`
`rwpFcv`

Because English is spoken at Palm, make sure the names are readable by developers who read English.

Naming Functions

By convention, Palm programmers name functions with mixed case with the first letter of each sub-word capitalized. Function names should include a verb to indicate the operation the function performs and a noun to indicate the data being manipulated. If you're designing a library, indicate private functions with the `Prv` prefix. Some examples of function names are `SetTimeOfNextAlarm`, and `PrvReadPreferences`.

Naming Variables

By convention, Palm programmers name variables with mixed case and with the first letter of each sub-word capitalized except for the very first letter of the name. Some examples are `addressRecordIndex` and `vehicleMakeName`.

A variable name can indicate important information about the variable. It is especially helpful to indicate that a variable is global. Global variables are tricky—if the variable gets changed unexpectedly, it can take some effort to determine exactly what part of your code changed it. Indicate your global variables with the letter “g”. Some developers put the “g” in the prefix, others put it in the suffix. In the Palm OS® source code, it most commonly appears in the prefix, for example, `gAddressRecordH`. If you’re modifying someone else’s code, use the convention in the code you’re modifying.

It’s also helpful to indicate if a variable is a pointer or a handle using the letters “P” and “H”, respectively. In the Palm OS source code, these most commonly appear in the suffix. If you’re modifying someone else’s code, use the convention in the code you’re modifying.

Naming Constants

By convention, Palm programmers name constants with fixed case and with the first letter of each sub-word capitalized except for the very first letter of the name. This naming convention is the same as the variable naming convention.

Avoid unnamed constants. Instead, give your constants readable and self-explanatory names. If the constant is used more than once, naming the constant simplifies the process of changing it. Naming the constant also give you a place to comment it. See “Documenting Constants” on page 12.

In very rare circumstances you might use a constant that has such an obvious value that naming it is superfluous. For example if you’re converting a number to a percentage, you don’t need to name the conversion factor. Everyone knows it’s 100. Another example is `numberOfSecondsInAMinute` in Listing 1.7 on page 12. If you’re thinking about using an unnamed constant, bear in mind that what may be obvious to you may not be obvious to other developers.

Recommendations

General Recommendations

- Choose names that are readable and self-explanatory to English readers.
- If you’re modifying code, stick to the naming conventions in the code you’re modifying.

Naming Functions

- Used mixed case for function names and capitalize the first letter of each sub-word.
- Begin private functions with `Prv`.

Naming Variables

- Use mixed case for variable names and capitalize the first letter of each sub-word except the very first letter of the name.
- Put the letter “g” in the prefix or suffix of the name to mark a global variable. In the Palm OS code, the “g” appears most commonly in the prefix.
- Put the letter “P” in the prefix or suffix of the name to mark a pointer. In the Palm OS code, the “P” appears most commonly in the suffix.
- Put the letter “H” in the prefix or suffix of the name to mark a handle. In the Palm OS code, the “H” appears most commonly in the suffix.

Naming Constants

- Use mixed case for constant names and capitalize the first letter of each sub-word except the very first letter of the name.
- Give your constants descriptive names.

Comments

Some people say that you need to have a particular type of comment header for every function. Others say that at least one third of your code should be comments. Regardless, everyone agrees that comments are extremely important to writing maintainable code.

Comments serve two very important purposes for the developer:

- They document what the code is supposed to do.
- They point out areas of the code that aren't completely obvious to a developer who is casually reading the code.

The following sections show how you can write comments that serve each of these purposes.

Comments Document What the Code Is Supposed to Do

When you debug another developer's code or your own code if you haven't touched it for a while, you don't always know exactly what the code is supposed to do. This can make it difficult to determine whether the code works correctly or not.

You tell other developers what your code is supposed to do by including comment headers in front of each function. These headers can be detailed or simple, long or short. But each comment header must have the following information:

- What is the purpose of the function?
- What are the parameters to the function and what do they mean?
- Are the parameters inputs, outputs, or both?
- Are there limits to the range of values for which the parameters are valid?
- What does the function return?
- If the function returns error codes, which error codes can it return, and what do the error codes mean?
- What assumptions does the function make about the state of the system, if any?

- What are the side-effects, if any, of calling the function?

With this information, the developer can quickly determine if the function performs its task correctly or not. If so, the developer can safely ignore the function. If not, the developer needs to investigate the code within the function.

Note that the comment header does not describe *how* the code works. It just describes *what* the code is supposed to do.

Comment Header Example

If you want to comment the `GetObjectPtr` function in Listing 1.3 on page 3, you could use a comment header in the format shown in Listing 1.4.

Listing 1.4 A function header for GetObjectPtr

```
// FUNCTION:
// GetObjectPtr
//
// DESCRIPTION:
// Returns a pointer to the user interface object
// with the specified ID. Assumes a form is active,
// the object ID is valid, and the object is in the
// currently active form.
//
// PARAMETERS:
// objectID - IN The ID of the desired object
//
// RETURNS:
// A pointer to the user interface object

void *GetObjectPtr(UInt16 objectID)
{
   FormPtr activeFormP = FrmGetActiveForm();
   return FrmGetObjectPtr(
      activeFormP,
      FrmGetObjectIndex(activeFormP,objectID));
}
```

This detailed comment header may seem like overkill, especially for such a simple function. The comment header in Listing 1.5 might be more appropriate.

Listing 1.5 A shorter function header for GetObjectPtr

```
// Returns a pointer to the user interface object
// with the specified ID. Assumes a form is active,
// the object ID is valid, and the object is in the
// currently active form.
```

Note that the comment header contains the same information a technical writer documents. It's also the same information a test engineer needs to test it.

Comments Point Out What Is Not Obvious

If you use good names and keep your code tidy, most of your code will be easy to understand. For example, the statement

```
activeFormP = FrmGetActiveForm();
```

needs no additional comments to explain what it does.

However, sometimes a single statement or a series of statements performs a task that isn't obvious when you first look at it. For example, consider Listing 1.6. Clearly the code sets an alarm, but where does the value of `minutesBetweenAlarms` come from? Why is `minutesBetweenAlarms` used twice in the call to `SetTimeOfNextAlarm`?

Listing 1.6 Example without comments

```
UInt32 minutesBetweenAlarms;
UInt32 currentTimeInSeconds;

minutesBetweenAlarms = ((SysAlarmTriggeredParamType *)
  commandParameterBlockP)->ref;
currentTimeInSeconds = TimGetSeconds();
SetTimeOfNextAlarm(currentTimeInSeconds +
  minutesBetweenAlarms * numberOfSecondsInAMinute,
  minutesBetweenAlarms);
```

Two comments answer these questions (see Listing 1.7). The comments imply that the code uses features of the alarm manager API that might be unfamiliar to the developer.

Listing 1.7 Example with comments

```
UInt32 minutesBetweenAlarms;
UInt32 currentTimeInSeconds;

// Get the time interval from the alarm’s caller-defined
// value
minutesBetweenAlarms = ((SysAlarmTriggeredParamType *)
   commandParameterBlockP)->ref;

// Schedule an alarm that triggers minutesBetweenAlarms
// minutes after the current system time and set the
// alarm’s caller-defined value to minutesBetweenAlarms
currentTimeInSeconds = TimGetSeconds();
SetTimeOfNextAlarm(currentTimeInSeconds +
   minutesBetweenAlarms * numberOfSecondsInAMinute,
   minutesBetweenAlarms);
```

Indicating Matching Braces

Another way that comments can point out what is not obvious appears in the last three lines of Listing 1.3 on page 3. Here, the comments indicate where the matching opening braces are. The reader doesn’t need to use a ruler to determine if the code is nested properly.

Documenting Constants

Use comments to document your constants. If you followed the recommendations in “Naming Constants” on page 7, you can document the constant where you define it. In particular, a constant that defines a limit or a size needs documentation that states the purpose of the constant and, if useful, how you chose its value.

```
// The default focal length of the camera lens in mm. Chosen
// to match that of a standard 35mm camera lens.
#define defaultLensFocalLength 45
```

Keeping Comments Up to Date

Whenever you change your code, be sure the comments pertain the new code. Whenever the comments do not reflect the actual code, they mislead the developer and cause him or her to question the accuracy of every comment in the code.

Recommendations

- Comments should be written for someone who reads English.
- Use comment headers to document what every function is supposed to do.
- Use comments to point out details that aren’t obvious to a developer who is casually reading your code.
- Use comments to indicate the matching open brace for every closing brace.
- When you define a constant, write a comment describing the purpose of the constant and how its value is chosen.
- Keep comments up to date. Obsolete comments are worse than none at all!

2

Handling Error Conditions

There are many situations where a program needs to handle error conditions; invalid user input, network failure, and full memory are just a few examples. Handling these error conditions can be difficult because you have to write code to check for error conditions, clean up allocated and locked memory chunks when errors occur, and handle the successful case when no errors occur. At the same time, you need to keep your code readable and easy to maintain.

This chapter presents a design pattern for handling errors that has proven to be both effective and easy to maintain.

Error Handling Issues

Two issues come up frequently in error handling code:

- Errors conditions are handled in a different place than where they are detected. Usually error conditions are displayed by the user interface but are detected by code nested several layers lower. Therefore, there must be an effective way to transfer the error information.
- Often when an error condition occurs within a function, memory chunks need to be deallocated and unlocked. This cleanup code should not be duplicated and yet it must account for all of the error conditions that could occur.

These issues and how to resolve them in a maintainable way are discussed in more detail in the following sections.

Propagating the Error

Typical Palm applications report error conditions using an alert in the user interface. The alert code generally resides in a function that handles an event that triggers some operation. The event handler typically calls a function to perform an operation. The code that first detects the error condition might be nested several layers beneath this function, and it needs to communicate the error condition to the event handler.

An example is an application that beams its information to another device. The event handler gets the event that the user wants to beam the data. It then calls a function to perform the actual beaming. That function might call some lower level functions that eventually call one or more Palm OS Exchange Manager functions. If there isn't enough memory to start the beaming process, the `ExcPut` function is the first to detect it. This information must then be communicated to the event handler.

A good way to propagate the error condition information from the inner function to the event handler that actually reports the error is as follows:

- Define unique error codes for all the error conditions your program will encounter. The Palm OS Error Manager defines several error codes including one that indicates no error has occurred.
- Define each function to return an error code.
- If a function detects an error condition, it should immediately clean up and return an appropriate error code.
- If a function calls another function that returns an error code indicating an error condition, the calling function should immediately clean up and return the same error code.

In the beaming example, when the low level `ExcPut` function fails to allocate the memory it needs, it returns an error code. The function that called it immediately cleans up and returns the same error code. In this way, the error code makes its way through layers of function calls until it reaches the event handler, which displays an alert.

Avoiding Duplicate Cleanup Code

As mentioned in the previous chapter, duplicate code is a maintenance headache and should be avoided. This is true for cleanup code that deallocates and unlocks memory chunks when an error occurs. A good approach for cleaning up is to design the cleanup code to handle all error condition cases including the success case. Put this code at the end of the function. Whenever an error condition is detected, control should transfer to this cleanup code via a `goto` statement.

The best way to show how this cleanup approach works is through an example, shown in Listing 2.1:

Listing 2.1 Example function returning an error code

```
Err SampleFunction(void)
{
  MemHandle someTypeH = NULL;
  SomeType *someTypeP = NULL;
  Err err = errNone;

  someTypeH = MemHandleNew(sizeof(SomeType));
  if (someTypeH == NULL) {
    err = memErrNotEnoughSpace;
    goto Done;
  }

  someTypeP = MemHandleLock(someTypeH);
  if (someTypeP == NULL) {
    err = memErrChunkNotLocked;
    goto Done;
  }

  // CODE
  // ...

  err = SomeFunctionThatUsesSomeType(someTypeP);
  if (err != errNone) goto Done;

  // CODE
  // ...

Done:
  if (someTypeP != NULL)
    MemPtrUnlock(someTypeP);
  if (someTypeH != NULL)
    MemHandleFree(someTypeH);

  return err;
}
```

At the beginning, the function defines local variables for each of the resources that will be allocated and sets these variables to `NULL`. This value indicates that the resources haven't been allocated yet.

When the function allocates the resources, it stores the value of the resource in the local variable. Because the variable is no longer `NULL`, the resource is marked to be cleaned up.

The cleanup code, which starts at the `Done` label, can handle the three error cases and the success case by unlocking the memory chunk only if it is successfully locked and freeing it only if it is successfully created. The four code paths are as follows:

- The memory chunk is not successfully created. In this case, `someTypeH` gets set to `NULL,` and `someTypeP` retains its initial value of `NULL`. The cleanup code does not attempt to unlock the chunk or free it.
- The memory chunk is created but not successfully locked. In this case, `someTypeH` is not `NULL`, but `someTypeP` is set to `NULL`. The cleanup code does not attempt to unlock the chunk, but it does free it.
- The memory chunk is created and locked, but the function that uses the pointer fails. In this case, both `someTypeH` and `someTypeP` are valid and not `NULL`. The cleanup code unlocks the chunk and frees it, and `SampleFunction` returns the error code generated by `SomeFunctionThatUsesSomeType`.
- The memory chunk is created, locked, and used successfully. Again, both `someTypeH` and `someTypeP` are valid and not `NULL`. The cleanup code unlocks the chunk and frees it, and `SampleFunction` returns the `errNone` error code, indicating that the function was successful.

The design pattern captured by this example effectively eliminates duplicate cleanup code. The memory chunk is only allocated in one place and deallocated in one place. Also, the chunk is only locked in one place and unlocked in one place. The arrangement has other advantages as well:

- You can add checks for more error conditions easily. Such a check only needs to set the error code and jump to `Done`. You don't have to worry about deallocating or unlocking memory chunks.

- The code is easy to read. The ‘successful’ stream of code is always left-justified. Only error cases are wrapped in `if` statements. Thus you never have to jump backwards or skip pieces of code. When you read the code, you know that at every point, all handles and pointers are valid.
- It propagates any error code to the function that called it.

The alternative, nesting the successful case within multiple `if` statements is hard to read—not only is the code highly indented but it also usually extends past the right side of the screen, making the code hard to navigate and read.

Other Considerations

Returning Allocated Data

If every function returns an error condition, you can’t pass a pointer or handle using the return value. Instead, pass it by reference using a parameter. The following code shows how to do this.

Listing 2.2 Returning allocated data example

```
Err SampleFunctionThatAllocates(SomeType **someTypePP)
{
  Err err = errNone;
  SomeType *someTypeP = NULL;

  // CODE THAT SETS someTypeP
  // ...

Done:
  if (err != errNone) {
    if (someTypeP != NULL) {
      MemPtrFree(someTypeP);
      someTypeP = NULL;
    }
  }

  // Clean up anything else

  if (someTypePP != NULL)
    *someTypePP = someTypeP;
```

```
   return err;
}
```

Unlocking a Memory Chunk Safely

If you need to unlock a memory chunk in the middle of a function, be sure to set the pointer that referenced the locked chunk to `NULL`. An example of when you might need to do this is resizing a chunk.

Listing 2.3 Unlocking a memory chunk example

```
// Unlock the chunk associated with someTypeP
MemPtrUnlock(someTypeP);
someTypeP = NULL;

// Now you can do something with the handle
```

This ensures that the cleanup code does not try to unlock the chunk again. In addition, if you accidentally write code that references the pointer after the chunk is unlocked, the code will attempt to access `NULL`, which is easier to detect than if it attempts to access a stale pointer.

Recommendation for Handling Errors

All of the material in this chapter is condensed into Listing 2.1 on page 17. This example propagates errors correctly up through nested function calls, cleans up all its resources, and is easy to read. If you follow this example whenever you write functions dealing with error conditions, your code will be simpler, more robust, and easier to maintain.